Мотивация и дизайн концептов с ограничениями подтипирования для языка программирования C#

Ю. В. Белякова

Научный руководитель: доцент каф. АДМ, к. ф.-м. н. С. С. Михалкович

Институт математики, механики и компьютерных наук им. И.И. Воровича
Южный федеральный университет

21 января 2015 г.

Содержание

Содержание

Определение

Обобщённое программирование [7]

Парадигма программирования, согласно которой алгоритмы и структуры данных определяются на *абстрактном* или *обобщённом* уровне, а не в терминах конкретных типов. Вместо конкретных типов используются *типовые параметры*.

- **Абстракции типов** описываются как множества требований к типам (например, операции над элементами типа).
- **Обобщённые алгоритмы и структуры данных** работают с типовыми параметрами, удовлетворяющими необходимым абстракциям.

Пример

Абстракция «Моноид»

Тип `T` — моноид, если выполнены требования:

1. Определена бинарная операция `binop(T, T): T`;
2. Определён нейтральный элемент `ident: T`.

Обобщённый алгоритм `accumulate` (псевдокод)

```
accumulate(T[] values): T [where T satisfies Monoid]
  var acc: T = ident
  foreach (var x: T in values)
    acc = binop(acc, x)
  return acc
```

Инстанции для конкретных типов

Целые числа (+, 0)

- binop = «+», ident = 0;
- accumulate — сумма чисел.

Целые числа (*, 1)

- binop = «*», ident = 1;
- accumulate — произведение чисел.

Строки (&&, "")

- binop = «&&» (конкатенация), ident = "" (пустая строка);
- accumulate — конкатенация строк.

Примечание. Примеры реализации на C++ и Haskell смотри в Приложении.

Содержание

Механизмы реализации средств обобщённого программирования

Существует два основных подхода к реализации средств обобщённого программирования:

1. **«Можно только то, что разрешено»**. В обобщённом коде доступны только те операции, которые явно прописаны в *ограничениях*.
   Примеры: C#, Java, Haskel, SML, ...
2. **«Можно всё, что не запрещено»**. Обобщённый код может содержать любые синтаксически корректные конструкции.
   Пример: C++.

Недостатки подходов

«Можно только то, что разрешено»

Возможный недостаток видов ограничений на типовые параметры, которые поддерживаются языком. Следствие — невозможность «комфортного» программирования [4].

«Можно всё, что не запрещено»

- Позднее обнаружение ошибок, сложные сообщения об ошибках [9, 3].
- Отсутствие модульности (шаблон хранится в виде синтаксического дерева, оно всякий раз используется для инстанцирования) [3].

Необходимые возможности механизма обобщённого программирования

В результате исследования [4] выделены характеристики, существенные для удобства обобщённого программирования на основе явных ограничений:

1. Ограничения на несколько типов.
2. Множественные ограничения.
3. Доступ к ассоциированным типам.
4. Ограничения на ассоциированные типы.
5. Возможность адаптации типа после его определения.
6. Раздельная компиляция обобщённых методов/типов и их инстанций.
7. Неявный вывод типов аргументов.

Поддержка в языках программирования

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●*	○	○	○	◒
Multiple constraints	-	◒	◒	●	○†	●	●	●
Associated type access	●	●	◒	●⋆	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

*Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).
⋆Using the functional dependencies extension to Haskell (Jones, 2000).
†Planned language additions.

Table 1: *The level of support for important properties for generic programming in the evaluated languages. A black circle indicates full support, a white circle indicates poor support, and a half-filled circle indicates partial support. The rating of "-" in the C++ column indicates that C++ does not explicitly support the feature, but one can still program as if the feature were supported due to the permissiveness of C++ templates.*

Рис. 1: Характеристики обобщённого программирования

Концепты

Термин **«концепт»** изначально использовался для неформального описания требований к типам-параметрам шаблонов C++ в документации к STL [1].

Позже в качестве замены неограниченных шаблонов было предложено ввести в C++ *новую конструкцию* — **концепты**.

- Работа над концептами ведётся с начала 2000-х [9, 3, 5, 10, 11].
- Высокий уровень выразительной силы, сравним с классами типов Haskell [2].
- Поддержка всех характеристик из таблицы на Рис. 1.

Концепты: терминология

Концепт Конструкция языка, задающая именованный набор требований (ограничений) к типам-параметрам концепта.

Модель Конструкция языка, задающая способ, которым тип (или типы) реализует концепт.

Концепты C++: пример

Моноид и accumulate

```
concept Monoid<T>
{
  T binOp(T x, T y);
  T ident();
}

template<T> requires Monoid<T>
T accumulate(T vals[],
             size_t cnt)
{
  T result = ident();
  for (int i = 0; i < cnt; ++i)
      result
      = binOp(result, vals[i]);
  return result;
}
```

```
model Monoid<int>
{
    int binOp(int x, int y)
    {
        return x + y;
    }
    int indent()
    {
        return 0;
    }
}

const size_t N = 5;
int values[N]{ 1, 2, 3, 4, };
int s = accumulate(values, N);
```

Концепты в языке G

	C++	SML	Haskell	Java	C#	$\mathcal{G}$
Multi-type concepts	-	●	●*	○	○	●
Multiple constraints	-	◒	●	●	●	●
Associated type access	●	●	◒	◒	◒	●
Constraints on assoc. types	-	●	●	◒	◒	●
Retroactive modeling	-	●	●	○	○	●
Type aliases	●	●	●	○	○	●
Separate compilation	○	●	●	●	●	●
Implicit instantiation	●	○	●	●	●	●
Concept dispatching	●	○	○	○	○	◒

*Using the multi-parameter type class extension to Haskell 98 [149].

Table 4.1: The level of support for generic programming in several languages. The rating of "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported due to the flexibility of C++ templates.

Рис. 2: Сравнение средств обобщённого программирования в языке G [8] (Джереми Сик)

Содержание

Дженерики в C#

- Механизм **явных ограничений**.
- Ограничения задаются с помощью **интерфейсов**.

Пример

```
interface IWeighed
{
    double GetWeight();
}

...
static double MaxWeight<T>(T[] values)
    where T : IWeighed
{
    double[] weights = values.Select(x => x.GetWeight());
    return weights.Max();
}
```

Невозможность адаптации типа (отсутствие retroactive modeling) I

Класс должен реализовывать интерфейс *при определении*.

```
class MyData
    : IWeighed                  // (!)
{
    ...
    double GetWeight() { ... }
    ...
}
```

Только в этом случае инстанция `MaxWeight<MyData>` корректна.

Невозможность адаптации типа (отсутствие retroactive modeling) II

Проблема

В случае класса `MyPoorData`

```
class MyPoorData
{
    ...
    double GetWeight() { ... }
    ...
}
```

инстанция `MaxWeight<MyPoorData>` невозможна.

Класс `MyPoorData` не реализует интерфейс `IWeighed`, хотя и *обладает соответствующей семантикой*.

Невозможность адаптации типа (отсутствие retroactive modeling) III

Что делать?

Использовать **паттерн Адаптер**.

- Определить класс-адаптер `MyWeighedData : IWeighed` класса `MyPoorData`.
- Сформировать массив `MyWeighedData[]` из массива `MyPoorData[]`.
- Использовать инстанцию `MaxWeight<MyWeighedData>` обобщённой функции `MaxWeight<T>`.

Решение проблемы в стандартной библиотеке .NET Framework

Для обеспечения «адаптации» типов реализованы две «ветви» перегруженных функций.

1. Для типов со встроенным интерфейсом.	2. Для типов с внешним классом-адаптером интерфейса.

`interface IComparable<T>`
`{ int CompareTo(T other); }`
 	
`Sort<T>(T[])`
`where T : IComparable<T>;`
	`interface IComparer<T>`
	`{ int Compare(T x, T y); }`
	
	`Sort<T>(T[], IComparer<T>);`

Пример класса с поддержкой адаптации типов

Бинарное дерево поиска элементов типа T

```
class BST<T> where T : IComparable<T>
{       ...
        public BST([args]) { ... }
        // lots of the code
}
class BSTRtrAct<T>
{       ...
        private IComparer<T> comparator;
        public BST([args,] IComparer<T> comparator) { ... }
        // lots of the code
}
```

Недостатки метода «классов-близнецов»

- Дублирование кода.
- Дополнительное поле `comparator` класса `BSTRtrAct<T>`.

Случай нескольких требований

Если типовый параметр некоторого обобщённого метода p<S> должен удовлетворять 2-м ограничениям, необходимо определить 4 перегруженных версии:

```
p<S>(...) where S : ISelf1<S>, ISelf2<S>;
p<S>(..., IExt1<S>) where S : ISelf2<S>;
p<S>(..., IExt2<S>) where S : ISelf1<S>;
p<S>(..., IExt1<S>, IExt2<S>);
```

Проблемы

- Количество перегрузок растёт экспоненциально.
- Интерфейс используется и как тип, и как ограничение.

Неоднозначное использование интерфейсов I

Интерфейс как ограничение

```
static double MaxWeight<T>(T[] values)
    where T : IWeighed
{ ... }
```

values — гомогенный массив элементов типа T, где T **ограничен** интерфейсом IWeighed.

Интерфейс как тип

```
static double Max(ICollection<double> values)
{ ... }
```

values — коллекция вещественных чисел;
ICollection<double> — **тип** всей коллекции.

Неоднозначное использование интерфейсов II

Интерфейс как тип

```
static bool ContainsGreater(IWeighed upper,
                            ICollection<IWeighed> values)
{
    return values.Any(x
        => x.GetWeight() > upper.GetWeight());
}
```

- upper — объект **типа** IWeighed;
- values — *гетерогенная* коллекция элементов **типа** IWeighed.

Неоднозначное использование интерфейсов III

Интерфейс как тип и ограничение

```
static bool ContainsGreater<T>(IWeighed upper,
                               ICollection<T> values)
    where T : IWeighed
{
    return values.Any(x
        => x.GetWeight() > upper.GetWeight());
}
```

- upper — объект **типа** IWeighed;
- values — *гомогенная* коллекция элементов типа T, где T **ограничен** интерфейсом IWeighed.

Пример интерфейса IComparable<T>

```
interface IComparable<T>
{ int CompareTo(T other); }

Sort<T>(T[]) where T : IComparable<T>
```

Стандартное использование интерфейса IComparable<T> предполагает, что элементы типа T можно сравнивать *друг с другом*.

Проблема бинарных методов

Определение интерфейса IComparable<T> не соответствует этой семантике.

Что делать с IComparable<T>?

Можно разбить интерфейс IComparable<T> на два различных интерфейса:

1. IComparableTo<S> (элементы типа, реализующего этот интерфейс, сравнимы с элементами типа S);
2. IComparable<T> (элементы типа T сравнимы друг с другом).

1

```
interface IComparableTo<S>
{ int CompareTo(S other); }

int Find<T, S>(T[] vals, S x)
    where T : IComparableTo<S>
{ ... }
```

2

```
interface IComparable<T>
    where T : IComparable<T> (*)
{ int CompareTo(T other); }

void Sort<T>(T[] vals)
    where T : IComparable<T> (*)
{ ... }
```

Пример графа I

`IDataGraph<Vertex, DataType>` — интерфейс графа особой структуры.

```
interface IDataVertex<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>          // (*)
{   ...
    IEnumerator<Vertex> OutVertices { get; }
    ...
}
interface IDataGraph<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>          // (#)
{ ... }
```

- В вершинах типа `Vertex` графа хранятся данные типа `DataType`;
- Вершина хранит ссылки на соседние вершины;
- Граф состоит из множества вершин, а не ребёр.

Пример графа II

Зачем нужны ограничения (*) и (#)?

```
class V1 : IDataVertex<V1, int> { ... }
class V2 : IDataVertex<V1, int> { ... }
```

Благодаря (*) и (#) инстанция `IDataGraph<V2, int>` недопустима.

Примечание. Типы `Vertex` вершины и `DataType` данных графа полностью определяются типом графа. `Vertex` и `DataType` — *ассоциированные* типы.

Распространение ограничений

В C# отсутствует *распространение ограничений*.

Ограничение на тип вершины графа определено в интерфейсе графа:

```
interface IDataGraph<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>          // (#)
{ ... }
```

Но всякий раз при использовании графа нужно указывать его (ограничение (*)):

```
G GetSubgraph<G, Vertex, DataType>(G g, Predicate<DataType> p)
    where G : IDataGraph<Vertex, DataType>, new()
    where Vertex : IDataVertex<Vertex, DataType>          // (*)
{ ... }
```

Паттерн Наблюдатель (Observer) I

В некоторых случаях нужно связать ограничениями несколько типовых параметров.

Пример

Паттерн Наблюдатель (Observer) [12].

- Связывает два типа.
- Каждый из этих типов работает с объектами другого типа из пары.

Паттерн Наблюдатель (Observer) II

Код на C#

IObserver<O, S> interface

```
interface IObserver<O, S>
    where O : IObserver<O, S>
    where S : ISubject<O, S>
{

    void update(S subj);

}
```

ISubject<O, S> interface

```
interface ISubject<O, S>
    where O : IObserver<O, S>
    where S : ISubject<O, S>
{
    List<O> getObservers();
    void register(O obs);
    void notify();
}
```

Два различных интерфейса с одинаковыми типовыми параметрами и ограничениями на них.

А что же с моноидом?

Проблема: неясно, что делать с нейтральным элементом

```
interface Monoid<T>
    where T : Monoid<T>
{
    T binop(T other);
    T ident();              // ???
}
...
static T Accumulate<T>(T[] values)
    where T : Monoid<T>
{
  T result = ???
  foreach (T val in values)
      result = result.binOp(val);
  return result;
}
```

Содержание

Модификации интерфейсов

Две работы, посвящённые модификации интерфейсов.

1. [6] (Jaakko Järvi et al. 2005 г.) расширяет интерфейсы C# ассоциированными типами и распространением ограничений.
2. [12] (Stefan Wehr et al. 2007 г.) представляет язык JavaGI — Java с генерализованными интерфейсами, допускающими адаптацию типов, ограничения на несколько типов.

Расширенные интерфейсы C# I

Преимущества

- Ассоциированные типы.
- Распространение ограничений.

Недостатки

- Невозможность адаптации типа.
- Не решается проблема ограничений на несколько типов.
- В некоторых случаях ещё более сложные «рекурсивные» ограничения на типовые параметры.

Расширенные интерфейсы C# II

```
interface IDataVertex<Vertex>
        where Vertex : IDataVertex<Vertex>
        where DataType == Vertex::DataType
{
    type DataType;
    ...
    IEnumerator<Vertex> OutVertices { get; }
    ...
}
interface IDataGraph
{   type Vertex : IDataVertex<Vertex>;
    type DataType;
    require DataType == Vertex::DataType;
    ...
}
```

```
static G GetSubgraph<G>(G g, Predicate<G::DataType> p)
     where G : IDataGraph, new()
{ ... }
```

JavaGI I

Преимущества

- Возможность адаптации типов.
- Ограничения на несколько типов.
- Решение проблемы бинарных методов.
- Статические методы.

Недостатки

- Отсутствие ассоциированных типов.
- **Интерфейсы больше нельзя использовать как типы** (в обычном ОО-смысле). Фактически, интерфейсы превратились в концепты.

JavaGI II

Паттерн Observer [12]

```
interface ObserverPattern[O, S] {
    receiver O { void update(S subj); }
    receiver S {
        List<O> getObservers();
        void register(O obs) { getObservers().add(obs); }
        void notify() {
            for (O obs : getObservers())
                obs.update(this);
        }
    }
}
class MultiheadedTest {
    <S,O> void genericUpdate(S subject, O observer)
    where [S,O] implements ObserverPattern {
        observer.update(subject);
    }
}
```

Тупик

Дилемма

- Проблемы адаптации типа и ограничений на несколько типов принципиально неразрешимы для существующих интерфейсов из мира ООП.
- Если заменить интерфейсы моделью интерфейсов JavaGI, то интерфейс больше не может использоваться как тип, он играет только роль ограничений.

Решение

Использовать для ограничения типовых параметров *другую конструкцию*, а за интерфейсами сохранить роль типов.

Интерфейсы как типы, концепты как ограничения

Имеет смысл ввести в язык новую конструкцию — **концепты**.

- Интерфейсы (в том числе обобщённые) сохраняют свою объектно-ориентированную сущность и используются в роли **типов**.
- Концепты используются в роли **ограничений** на типовые параметры обобщённого кода.

Пример использования обобщённых интерфейсов

```
interface IFunction<V> { V Apply(V x); }

static V ApplyComposition<V>(V initVal, List<IFunction<V>> funcs)
{
        V result = initVal;
        foreach (var func in funcs)
                result = func.Apply(result);
        return result;
}

static void TestFuncsComposition()
{
        List<IFunction<int>> intFuncs =
                new List<IFunction<int>>();
        // fill in the intFuncs: (* 2), (/ 4)

        int y = ApplyComposition(6, intFuncs);
        Debug.Assert(y == 3);  // 3 * 2 / 4 == 3
}
```

Пример использования концептов

```
concept CSemigroup[T] { T BinOp(T x, T y); }

model CSemigroup[int] // BinOp for int type is addition (+)
{ int BinOp(int x, int y){ return x + y; } }

static T Fold<T>(T initValue, List<T> values)
    where CSemigroup[T]
{
    T result = initValue;
    foreach (var val in values)
            result = BinOp(result, val);
    return result;
}

static void TestFold
{
    List<int> values = new List<int>(new int[]{ 3, 5, -1, 2 });
    int y = Fold(0, values);
    Debug.Assert(y == 9); // 0 + 3 + 5 + (-1) + 2 == 9
}
```

Содержание

Основные характеристики

Дизайн концептов поддерживает все основные характеристики из таблицы на Рис. 1. В том числе адаптацию типа, ассоциированные типы, ограничения на несколько типов, уточнение концептов.

Отличительные особенности концептов C#

- Ограничения подтипирования и надтипирования.
- Анонимные модели.

Базовые возможности

```
concept CEquatible[T]
{
        bool Equal(T x, T y);     // сигнатура функции
        // сигнатура функции с реализацией по умолчанию
        bool NotEqual(T x, T y)
        { return !Equal(x, y); }
}
// уточняющий концепт
concept CComparable[T] refines CEquatible[T]
{
    int Compare(T x, T y);
    // переопределяет Equal уточнённого концепта CEquatible[T]
    override bool Equal(T x, T y)
    { return Compare(x, y) == 0; }
}

concept CSemilattice[T] refines CEquatible[T]
{ T Join(T x, T y); }
concept CBoundedSemilattice[T] refines CSemilattice[T]
{       T Top;    }      // ассоциированное значение
```

Ассоциированные типы и вложенные концепт-требования

```
concept CTransferFunction[TF]
{
    type Data;                    // ассоциированный тип
    // вложенное концепт требование
    require CSemilattice[Data];

    Data Apply(TF trFun, Data d);
    TF Compose(TF trFun1, TF trFun2);
}

concept CDataVertex[Vertex]
{
    type Data;                    // ассоциированный тип

    Data GetValue(Vertex v);   // сигнатура функции
    void AddInVertex(Vertex vert, Vertex inVert);

    IEnumerator<Vertex> GetInVerticesEnumerator(Vertex vert);
    ...
}
```

Алиасы и явные концепты

```
concept CDataGraph[Graph]
{
    type Vertex;                    // ассоциированный тип
    // вложенное концепт требование с алиасом
    require CDataVertex[Vertex] using cVert; // то же, что:
        // require CDataVertex[Vertex];
        // using cVert = CDataVertex[Vertex];

    // ассоциированный тип с ограничением равенства
    type Data == cVert.Data;   // то же, что:
        // type Data;
        // require Data == cVert.Data;
    IEnumerator<Vertex> GetVerticesEnumerator(Graph g);
    ...
}

// явный концепт
explicit concept CBasicBlockVertex[Vertex]
    : refines CDataVertex[Vertex] {}
```

Ограничения на несколько типов

```
concept CObserverPattern[O, S]
{
    void UpdateSubject(O obs, S subj);

    ICollection<O> GetObservers(S subj);

    void RegisterObserver(S subj, O obs)
    { GetObservers(subj).Add(obs); }

    void NotifyObservers(S subj)
    {
        foreach (O obs in GetObservers(subj))
            UpdateSubject(obs, subj);
    }
}

class MultiheadedTest
{
    void GenericUpdate<S, O>(S subject, O observer)
        where CObserverPattern[S, O]
    { UpdateSubject(observer, subject); }

    void CallGenericUpdate()
    { GenericUpdate(new Model(), new Display()); }
}
```

Обобщённые методы

```
static void Sort<T>(T[] values)
    where CComparable[T]
{ ... }
```

```
bool CheckGraph<G>(G graph, Predicate<cGr.Data> pred)
    where CDataGraph[G] using cGr
    // алиас для концепт требования
    using cVert = CDataVertex[cGr.Vertex]
{
    ...
    var outVertices = cVert.GetOutVerticesEnumerator(...);
    ...
}
```

Обобщённые классы

```
class BinarySearchTree<T>
        // концепт требование с алиасом
        where CComparable[T] using cCmp
{
    private BinTreeNode<T> root;
    ...
    private bool AddAux(T x, ref BinTreeNode<T> root)
    {   ...
        // обращение к концепту по алиасу
        if (cCmp.Equal(x, root.data))
                return false;
        // функция Compare(,) из концепта CComparable[T]
        else if (Compare(x, root.data) < 0)
            return AddAux(x, ref root.left);
        ...
}

class BSTSet<T>
{
    // ограничение CComparable[T] распространяется
    private BinarySearchTree<T> bst;
    ...
    public bool FindMaxLower(T x, out T maxLow)
    {   ...
        // обращение к концепту через ключевое слово 'reqs'
        if (reqs.Comparable[T].Compare(curr.data, x) < 0)
        ...
    }
}
```

Пример с моноидом

```
concept CMonoid[T]
{
    T binOp(T x, T y);
    T ident;
}

static T Accumulate<T>(T[] values)
    where CMonoid[T] using cM
{
    T result = cM.ident;
    foreach (T val in values)
        result = cM.binOp(result, val);
    return result;
}
```

Ограничения подтипирования I

```
class PlainObjectData { ... }

[Serializable]
abstract class SecureContainer<T> : ISerializable
{
    ...
    public void Init(T value, CipherKey key)
    {
        PlainObjectData data = this.Serialize(value);
        // ... encryption ...
    }
    public T Release(DecipherKey key) { ... }
    ...
    protected abstract PlainObjectData Serialize(T value);
    protected abstract T Deserialize(PlainObjectData data);
}
```

Ограничения подтипирования II

```
concept CSecureSerializable[T]
{
    type Container <: SecureContainer<T>;
    require Container : new();
}
...
static void SecureSerialize<T>(T value, CipherKey key,
            string fileName, IFormatter formatter)
where CSecureSerializable[T] using cS
{
    FileStream fs = new FileStream(fileName, ...);
    cS.Container secureContainer = new cS.Container();
    secureContainer.Init(value, key);
    formatter.Serialize(fs, secureContainer);
    fs.Close();
}
static T SecureDeserialize<T>(DecipherKey key,
        string fileName, IFormatter formatter)
where CSecureSerializable[T] using cS { ... }
```

Определение модели

```
class Rational
{
    ...
    public int Num { get { ... } }
    public int Denom { get { ... } }
    public Rational Add(Rational other) { ... }
    ...
}

model CComparable[Rational]
{
    bool Equal(Rational x, Rational y)
    { return (x.Num == y.Num) && (x.Denom == y.Denom); }

    int Compare(Rational x, Rational y) { ... }
}


...
    var rationals = new BSTSet<Rational>();
    rationals.Add(new Rational(-5, 17));
```

Использование анонимной модели

```
static bool Contains<T>(T x, IEnumerable<T> values)
    where CEquatible[T]
{ ... }

static void TestContains
{
    Rational[] nums = ...;
    var hasNumer5 = Contains[model CEquatible[Rational] {
        bool Equal(Rational x, Rational y)
        { return x.Num == y.Num; }
    }](new Rational(5), nums);
}
```

Содержание

Общая концепция перевода

Результат перевода концептов и обобщённых конструкций — обобщённый код. Результирующий код насыщен мета-информацией в форме атрибутов.

Перевод основных конструкций.

- Концепт — абстрактный дженерик-класс. Типовые параметры и ассоциированные типы текущего концепта, а также вложенные концепты и их типы — типовые параметры обобщённого класса.
- Модель — наследник сконструированного абстрактного дженерик-класса концепта.
- Объекты моделей — синглтоны.
- ...

Отображение элементов расширенного языка

Конструкция расширенного языка	**Конструкция базового языка**
Концепт	Абстрактный класс
Параметр концепта	Типовый параметр
Ассоциированный тип	Типовый параметр
Уточнение концепта	Подтипирование
Ассоциированное значение	Свойство (только на чтение)
Вложенное концепт-требование	Типовый параметр
Концепт-требование в обобщ. коде	Типовый параметр
Модель	Класс

Трансляция концептов: простые примеры

```
[Concept]
abstract class CEquatible<[IsConceptParam]T>
{
    public abstract bool Equal(T x, T y);
    public virtual bool NotEqual(T x, T y)
    { return !this.Equal(x, y); }
}

[Concept]
abstract class CComparable<[IsConceptParam]T> : CEquatible<T>
{
    public abstract int Compare(T x, T y);
    public override bool Equal(T x, T y)
    { return Compare(x, y) == 0; }
}

[Concept]
abstract class CTransferFunction<[IsConceptParam]TF, [IsAssocType]Data,
                [IsNestedConceptReq]CSemilattice_Data>
    where CSemilattice_Data : CSemilattice<Data>, new()
{
    public abstract Data Apply(TF trFun, Data d);
    public abstract TF Compose(TF trFun1, TF trFun2);
}
```

Код на расширенном C# смотри на Сл. 47, Сл. 48.

Трансляция обобщённого кода

```
[GenericClass]
[ConceptAlias("CComparable_T", "cCmp")]
class BinarySearchTree<[IsGenericParam]T, [IsRequireConceptParam]CComparable_T>
    where CComparable_T : CComparable<T>, new()
{
    private BinTreeNode<T> root;
    ...
    private bool AddAux(T x, ref BinTreeNode<T> root)
    {
        ...
        CComparable_T cCmp = ConceptSingleton<CComparable_T>.Instance;
        // обращение к концепту по алиасу
        if (cCmp.Equal(x, root.data))
            return false;
        // функция Compare(,) из концепта CComparable[T]
        else if (ConceptSingleton<CComparable_T>.Instance
                 .Compare(x, root.data) < 0)
            return AddAux(x, ref root.left);
        ...
}
```

Код на расширенном C# смотри на Сл. 52.

Класс-синглтон: объекты концептов и моделей

```
public static class ConceptSingleton<T>
    where T : class, new()
{
    private static readonly T instance = new T();

    public static T Instance
    {
        get { return instance; }
    }
}
```

Трансляция моделей и инстанцирования

```
[ExplicitModel]
class MComparable_Rational : CComparable<Rational>
{
    public override bool Equal(Rational x, Rational y)
    { return (x.Num == y.Num) && (x.Denom == y.Denom); }

    public override int Compare(Rational x, Rational y) { ... }
}
```

Вызов обобщённого метода

```
// C# с концептами
BSTSet<Rational> rations = new BSTSet<Rational>();

// Базовый C#
BSTSet<Rational, MComparable_Rational> rations
    = new BSTSet<Rational, MComparable_Rational>();
```

Содержание

Дизайн концептов: заключение

В таблице на Рис. 3 представлено сравнение дизайнов концептов и модифицированных интерфейсов (ext. C# соответствует C# с расширенными интерфейсами, колонка $C\#^{Cpt}$ представляет дизайн C# с концептами).

Характеристика	**G**	**C++**	**ext. C#**	**JavaGI**	**$C\#^{Cpt}$**
ограничения на несколько типов	+	+	−	−	+
ассоциированные типы	+	+	+	−	+
ограничения равенства типов	+	+	+	−	+
ограничения подтипирования	−	−	+	+	+
адаптация типов	+	+	−	+	+
множественные модели	+	−	−	−	+
анонимные модели	−	−	−	−	+
перегрузка на основе концептов	+	+	−	−	−
раздельная компиляция	+	+	+	+	+

Рис. 3: Сравнение дизайнов “концептов”

Особенности трансляции

- Результирующий код — обобщённый. Концепты переводятся в дополнительные типовые параметры, а не дополнительные поля и параметры функций (в отличие от G [8] и JavaGI [12]). За счёт этого дополнительные расходы ложатся на этап компиляции, а не выполнения.
- MSIL-код хранит обобщённый код. Поэтому возможно восстановление кода на C# с концептами.
- Широко применяется метаинформация в форме атрибутов.

Дальнейшая работа

- Разработка модельного компилятора C# с концептами (трансляция в базовый язык).
- Доказательство типобезопасности расширения концептов (расширение FGJ концептами, трансляция в FGJ, доказательство корректности трансляции).

Литература I

[1] Matthew H. Austern.
Generic Programming and the STL: Using and Extending the C++ Standard Template Library.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz.
A comparison of c++ concepts and haskell type classes.
In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 37–48, New York, NY, USA, 2008. ACM.

[3] Gabriel Dos Reis and Bjarne Stroustrup.
Specifying c++ concepts.
In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 295–308, New York, NY, USA, 2006. ACM.

[4] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock.
An extended comparative study of language support for generic programming.
J. Funct. Program., 17(2):145–205, March 2007.

Литература II

[5] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine.
Concepts: Linguistic support for generic programming in c++.
In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[6] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine.
Associated types and constraint propagation for mainstream object-oriented generics.
In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 1–19, New York, NY, USA, 2005. ACM.

[7] DavidR. Musser and AlexanderA. Stepanov.
Generic programming.
In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 1989.

[8] Jeremy G. Siek.
A Language for Generic Programming.
PhD thesis, Indianapolis, IN, USA, 2005.
AAI3183499.

Литература III

[9] Bjarne Stroustrup and Gabriel Dos Reis.
Concepts — design choices for template argument checking.
Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, October 2003.

[10] Bjarne Stroustrup, Gabriel Dos Reis, and Alisdair Meredith.
Axioms: Semantics aspects of c++ concepts.
Technical Report N2887=09-0077, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, June 2009.

[11] Bjarne Stroustrup and Andrew Sutton.
A concept design for the stl.
Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, January 2012.

[12] Stefan Wehr, Ralf Lämmel, and Peter Thiemann.
Javagi: Generalized interfaces for java.
In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer Berlin Heidelberg, 2007.

Содержание

Примеры на разных ЯП I

```
template <typename T>
struct Monoid {
    static const T ident;
};

template<typename T>
T accumulate(T values[], size_t count)
{
    T acc = ident();
    for (size_t i = 0; i < count; ++i)
        acc = binop(acc, values[i]);
    return acc;
}
```

Листинг 1: Язык C++: обобщённый код

Примеры на разных ЯП II

```
template <>
struct Monoid<int> {
    static const int ident = 0;
};

int binop(int x, int y)
{
    return x + y;
}

int main()
{
    size_t const N = 3;
    int a[N]{ 1, 3, 5 };
    std::cout << accumulate(a, N); // 9
}
```

Листинг 2: Язык C++: инстанцирование

Примеры на разных ЯП III

```haskell
class Monid a where
    ident :: a
    binop :: a -> a -> a

accumulate :: (Monoid a) => [a] -> a
accumulate [] = ident
accumulate [val] = val
accumulate (val:values) = binop val (accumulate values)

instance Monoid Int where
    ident = 0
    binop = (+)

main = print $ accumulate ([1, 3, 5] :: [Int]) -- 9
```

Листинг 3: Язык Haskell